National Security Agency/Central Security Service

Information Assurance Directorate

Reducing the Effectiveness of Pass-the-Hash

March 19, 2013

A product of the Network Components and Applications Division

TSA-13-1005-SG

Contents

List of Figures

Disclaimer

This Guide is provided "as is." Any express or implied warranties, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the United States Government be liable for any direct, indirect, incidental, special, exemplary or consequential damages (including, but not limited to, procurement of substitute goods or services, loss of use, data or profits, or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this Guide, even if advised of the possibility of such damage.

The User of this Guide agrees to hold harmless and indemnify the United States Government, its agents and employees from every claim or liability (whether in tort or in contract), including attorneys' fees, court costs, and expenses, arising in direct consequence of Recipient's use of the item, including, but not limited to, claims or liabilities made for injury to or death of personnel of User or third parties, damage to or destruction of property of User or third parties, and infringement or other violations of intellectual property or technical data rights.

Nothing in this Guide is intended to constitute an endorsement, explicit or implied, by the U.S. Government of any particular manufacturer's product or service.

Trademark Information

This publication has not been authorized, sponsored, or otherwise approved by Microsoft Corporation.

Microsoft®, Windows®, Windows Server®, Windows Vista®, Active Directory®, Windows PowerShell ™, and Windows® Firewall with Advanced Security are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries.

1 Introduction

As network defenders implement new capabilities to protect their resources, attackers continue to discover ways to accomplish their objectives. While initial attack vectors constantly change with each newly applied security patch, some techniques for compromising additional machines, after initial exploitation, remain the same. Pass-the-Hash (PtH) is one of these techniques. In December 2012, Microsoft released a whitepaper[1] which discusses PtH in-depth, identifies numerous risk factors that make an organization highly vulnerable to PtH, and describes several mitigations. The purpose of this document is to expand on the ideas presented in [1] and provide guidance to DoD administrators.

PtH is not a new technique. The concept of PtH was publicly released in 1997[2]. A PtH attack reuses login credentials on one computer to access another computer with equivalent credentials. PtH is a common and prevalent technique that helps attackers spread laterally across a target network with ease.

The success of a PtH attack is dependent on many factors including the network configuration, firewall settings, account settings, etc. To defeat PtH attacks, administrators must differentiate authorized and unauthorized uses of Single Sign On (SSO), which is extremely difficult or impractical within any contemporary SSO architecture. SSO allows users to authenticate with the operating system *once* (generally once per login) and caches the user's credentials in memory. Each subsequent time the system requires the user's credentials to perform an action, instead of re-prompting the user, the cached credentials are used. The basic mechanisms on which PtH exploits cannot be immediately fixed with a patch and will require a fundamental SSO architecture design change. This document discusses mitigations administrators can deploy, in the interim, to reduce PtH's effectiveness by addressing some of the properties it depends upon.

2 Background

Passwords remain a common form of user authentication to computer systems. For each user account, the system stores a username and a representation of the corresponding password. The operating system generally uses cryptographic hash functions to store the hash of the password, instead of storing the password in cleartext. A hash value, or simply a hash, is the output of a hash function. Hash functions are one-way mathematical functions that take an input string and produce some fixed-size and deterministic output string. When a user is prompted for their password during authentication, the system takes the password, computes the hash value, and compares the computed hash against the stored hash. If the hashes are equivalent, then the user is authenticated and is allowed access to the system; otherwise, the authentication fails. Different systems and services allow users to authenticate in different ways. Most require a password, while others allow users to authenticate with a password hash.

A PtH attack harvests hashed user credentials and reuses them to authenticate with services supporting a password hash as an authenticator. An attacker with administrative privileges can retrieve all of the

hashed user credentials from disk by reading the Security Account Manager (SAM) file or by reading the Local Security Authority Subsystem Service (LSASS) process's memory. These credentials, if identical to credentials accepted on another machine, can be used to authenticate with remote Windows services and gain access to those systems.

Security policies continue to advocate stronger passwords that combine upper and lower case letters, special characters, and numbers. Longer, more complex passwords typically increase the difficulty of password cracking. However, password strength does not affect the success of PtH. This increases PtH's potential value in environments implementing strong password requirements.

The following attack describes a portion of the PtH methodology to gain domain administrator privileges. First, the attacker compromises an initial machine via a remote exploit or a client-side attack. Depending on the exploit, the attacker may only have user privileges and must escalate privileges to an administrator or system account, which grants access to all credentials on the local system. Next, the attacker attempts to use each set of credentials (preferably administrator credentials) to authenticate with other systems on the network via PtH. If any credentials grant access to a new machine, then more credentials are harvested. The attacker continues to spread across the network by employing this methodology and potentially gaining more credentials on each newly exploited machine until the domain administrator account is obtained or until all credentials have been exhausted. At which point, the attacker may have to use more difficult and noisier methods to expand control increasing the likelihood of being discovered.

3 Mitigations

The availability of free tools for obtaining user credentials continues to increase, e.g., Metasploit, Windows Credential Editor[3] (WCE), and Mimikatz[4], giving attackers a trivial way to increase their presence on a network. Successfully launching a PtH attack depends on the following conditions:

- The credentials used in the PtH attack must be valid credentials on the target system.
- The associated account name from the compromised credentials must have the Network Logon right on the target.
- The source of the attack must be able to communicate with the target over the network and the target must be configured to accept network connections.

These are the fundamental properties this paper's mitigations address.

Section 3.1 discusses the complications arising from duplicating credentials across machines. Section 3.2 discusses restricting local administrator accounts from remotely accessing systems. Section 3.3 provides Windows Firewall rules for restricting lateral movement between workstations. Deploying at least one of these mitigations is recommended, but implementing more will increase overall network security.

3.1 Creating unique local account passwords

A common practice for deploying an enterprise network is creating an image for a base machine and using this image as the baseline for many systems on the network. A side effect of this deployment

strategy is that all of the built-in administrator accounts, across all machines, will have the same password. Recall the attack from Section 2. The adversary, on each machine on the network, wants to harvest all user credentials from the machine and reuse them to potentially authenticate to other systems. Under this scenario, the attacker will trivially enumerate all of the systems on the network via PtH because all local administrator credentials are the same. Even worse, when gaining unauthorized access to the new machine, the attacker will already possess administrator privileges allowing them to immediately harvest more credentials. Unfortunately, this common setup is the ideal environment for PtH.

Enforcing unique local administrator credentials on each machine forces the adversary to perform additional steps to achieve their goal. For example, the attacker may be forced to crack the administrator's password offline, use PtH with a standard user's account, or exploit the machine (which may make detecting an ongoing attack easier). Appendix A contains a PowerShell script that enforces unique local credentials by creating a different random password for a specified user on each machine in a domain. The script outputs two text files. The first file contains a summary of successful password changes with the new password for each computer and the second file lists the machines with unsuccessful password changes, e.g., a machine is powered off. By default, the script forces the password file to be saved to a USB drive rather than the local file system, where it can be removed and not accessed by a remote adversary. Using an encrypted removable USB drive to store this data is recommended.

3.2 Denying local accounts from network logons

The mitigation in Section 3.1 curbs an adversary from performing PtH attacks using local administrator accounts in networks with duplicated passwords. This section presents a mitigation for restricting remote access to local administrator accounts.

Windows login architecture uses many different logon types. Each logon type is associated with a set of connection methods, which store credentials differently. Common logon types include: **interactive**, **remote interactive,** and **network**. **Interactive** logons, generally associated with a console logon or the RUNAS command, leave hashes in memory. **Remote interactive** logons also leave credentials in memory and are associated with Remote Desktop connections. N**etwork** logons through commands such as **psexec** (without the –u option) **net use**, and **MMC snap-ins,** however, do not leave hashes in memory. See Table 6 in [1] for a list of common remote connection methods and where hashes are stored, if anywhere.

Local, non-service accounts do not generally require remote login privileges in a domain setting to perform their required tasks. Therefore, removing the **network** and **remote interactive** logon privileges from these accounts, especially local administrator accounts, will harden the system and prevent an attacker from using PtH to obtain unauthorized access to new machines. Denying local administrators remote access forces machines to be physically administered or remotely administered through a domain account. Physically administering a machine is the most secure method, but this may be an unrealistic administration method for some networks.

Remotely administering machines with a domain account provides convenience, but, if not done securely, can leave systems vulnerable to PtH attacks. First, when possible, machines should be administered with tools and methods that do not leave reusable credentials in memory (see Table 6 in [1] for a list of common methods). Second, domain accounts should be used in a way that conforms to the principles of least privilege and system isolation. That is, systems should be administered with the least privileges possible to perform the necessary task and highly trusted accounts should not administer lower trusted workstations. For example, the domain administrator account should be used to administer the domain controller, but not user workstations. This way, if a lower trusted system is compromised, then the attacker is limited to only compromising other lower trusted systems.

The script in Appendix B runs locally as a startup script on each system in a domain and puts all local administrator accounts into a well-defined local group. The domain administrator can configure the **Deny access to this computer from the network** and the **Deny log on through Remote Desktop Services** (for Windows Server 2008R2 and later) policies to include the local group created by the script, which blocks local administrators from remotely logging in and also creates a tripwire[5]. If a local administrator account tries to remotely log in, e.g., an adversary in a PtH attack, then the system can be configured to trigger an alert for further investigation by an administrator.

3.3 Restricting lateral movement on the network with firewall rules

To successfully perform a PtH attack, the source of the attack must be able to communicate with the target. Therefore, restrict workstations from communicating directly with other workstations using Windows Firewall rules. This should have little impact on the user since workstations generally have no need to communicate with each other. If a workstation has services which require other workstations to communicate with it, the firewall can be configured to only allow that specific traffic through. Implementing this mitigation correctly requires an administrator to thoroughly understand the entire network and provided services, but also provides a larger improvement to a network's security posture. It also reduces the number of potential targets for a PtH attack more than the mitigations presented in Sections 3.1 and 3.2. After implementing these firewall rules, an adversary on a workstation may only attack systems not protected by the firewall, find an exploit to evade the firewall, or attack servers which should have a smaller attack surface and are monitored more closely.

Appendix C provides step-by-step instructions for setting up Windows firewall rules through Group Policy Object (GPO) settings, which disallow workstation to workstation traffic.

4 Conclusion

This document expands on the ideas presented by [1] and provides additional guidance to DoD administrators, including scripts and walkthroughs, for implementing three PtH mitigations. To reduce the likelihood of adversaries easily spreading across a network, deploying at least one of the discussed PtH mitigations is recommended. Deploying multiple mitigations will further increase overall network security.

5 References

[1] Mitigating Pass-the-Hash (PtH) Attacks and Other Credential Theft Techniques. December 2012. http://www.microsoft.com/en-us/download/details.aspx?id=36036.

[2] Bugtraq mailing list. NT "Pass the Hash" with modified SMB client vulnerability. April 8, 1997. http://www.securityfocus.com/bind/233.

[3] Windows Credential Editor website. http://www.ampliasecurity.com/research.html.

[4] Mimikatz website. http://www.pentestmonkey.net/blog/mimikatz-tool-to-recover-cleartext-passwords-from-lsass.

[5] Spotting the Adversary with Windows Event Log Monitoring. February 2013. http://www.nsa.gov/ia/mitigation_guidance

[6] Microsoft network port requirements website. http://support.microsoft.com/kb/832017

Appendix A: Creating unique local passwords

This script requires Powershell version 2 or later and changes the local account password for all specified machines. The length of the password is configurable by the **minPasswordLength** and **maxPasswordLength** parameters. The machine names to enumerate is defined by the **machinesFilePath** variable or, by default, all of the systems registered on the domain are used, except for domain controllers, which do not contain local accounts. If the **machinesFilePath** switch is used, then the program expects a file listing the hostnames with one hostname on each line.

Passwords are generated using the RNGCryptoServiceProvider .Net object. The local account (denoted by **localAccountName**) password is changed via the IADsUser interface. If for any reason the password change is unsuccessful, then the program will notify the administrator which machine names failed. At the end, the script will write a tab delimited file to the path of **outFilePath** with each line consisting of: **machineName localAccountName newPassword** and a separate file containing a list of each machine, one on each line, which did not have the password changed successfully.

By default, the output is saved to a USB drive, but this can be disabled by setting **forceUSBKeyUsage** to **$FALSE**. Running this script with the least privileges possible is recommended.

Example usages:

Run script with automatic machine detection with output to USB drive.
.\changeAccountPasswordOverNetwork -localAccountName Administrator -outFilePath D:\passwords.out

Run script with user-defined machine list, output to the local hard drive, and password length between 15 and 30 characters.

.\changeAccountPasswordOverNetwork –machinesFilePath C:\machines.txt -localAccountName Administrator -outFilePath C:\passwords.out -forceUSBKeyUsage $FALSE -minPasswordLength 15 -maxPasswordLength 30

```
param(
          [Parameter(Mandatory=$FALSE)][string]$machinesFilePath = "",
          [Parameter(Mandatory=$FALSE)][int]$minPasswordLength = 14,
          [Parameter(Mandatory=$FALSE)][int]$maxPasswordLength = 25,
          [Parameter(Mandatory=$FALSE)][int]$maxThreads = 20,
          [Parameter(Mandatory=$FALSE)][bool]$forceUSBKeyUsage = $TRUE,
          [Parameter(Mandatory=$TRUE)][string]$localAccountName,
          [Parameter(Mandatory=$TRUE)][string]$outFilePath
)

function Get-DomainComputersSansDomainControllers(){
          #Write every computer name in the domain minus the domain controllers
          #to the pipeline
          try{
                    $objDomain = New-Object System.DirectoryServices.DirectoryEntry

                    $objSearcher = New-Object System.DirectoryServices.DirectorySearcher
                    $objSearcher.SearchRoot = $objDomain
                    #get all computers that are not domain controllers
                    #the primary group of a DC cannot be changed
                    $objSearcher.Filter = ("(&(objectCategory=computer)(!(primaryGroupID=516)))")
                    $objSearcher.PropertiesToLoad.Add("name") > $NULL

                    $colResults = $objSearcher.FindAll()
                    foreach($objResult in $colResults){
                              $objComputer = $objResult.Properties
                              Write-Output $objComputer.name
                    }
          }
          finally{$objDomain = $NULL; $objSearcher = $NULL}
}

function SecureStringToString($secureStr){
          #convert System.Security.SecureString to String
    return $([Runtime.InteropServices.Marshal]::PtrToStringAuto([Runtime.InteropServices.Marshal]::SecureStringToBStr($secureStr)))
}

function readInFile($filePath){
          #read in each line of a file and output it to the pipeline
          $filePath = Resolve-Path $filePath
          if((Test-Path $filePath) -eq $FALSE){
                    [console]::WriteLine("File $filePath does not exist")
                    throw "The machine file listed at $filePath does not exist"
          }
          $reader = [System.IO.File]::OpenText($filePath)
          try{
                    for(;;){
                              $line = $reader.ReadLine()
                              if($line -eq $null){
                                        break
                              }
                              Write-Output $line
                    }
          }
          finally{$reader.Close();$reader = $NULL}
}

function genPassword($passwordLength){
          #ascii range (inclusive) for lower,upper,digits,and special chars
          $minAscii = 33
          $maxAscii = 126

          $containsLower = $containsUpper = $containsDigit = $containsSpecial = $FALSE
          [byte[]]$byte = New-Object Byte[] 1
          $password = New-Object System.Security.SecureString
          $random = New-Object System.Security.Cryptography.RNGCryptoServiceProvider
          try{
                    do{
                              #loop as long as the password length < the desired password length or
                              #the password does not conform to the password policy, which states
                              #a password contains lower, upper, digits, and special chars. Add
                              #1 byte to the SecureString on each iteration iff it is in the desired ascii range.
                              if($password.length -ge $passwordLength){
                                        #password failed password policy restrictions
                                        $password.clear()
                                        $containsLower = $containsUpper = $containsDigit = $containsSpecial = $FALSE
```

```
                                }

                                $random.getBytes($byte)
                                if(($byte[0] -ge $minAscii) -and ($byte[0] -le $maxAscii)){
                                          $password.appendChar([char]$byte[0])
                                          if([char]::IsLower($byte[0])){
                                                    $containsLower = $TRUE
                                          }
                                          elseif([char]::IsUpper($byte[0])){
                                                    $containsUpper = $TRUE
                                          }
                                          elseif([char]::IsDigit($byte[0])){
                                                    $containsDigit = $TRUE
                                          }
                                          elseif(([char]::IsSymbol($byte[0])) -or ([char]::IsPunctuation($byte[0]))){
                                                    $containsSpecial = $TRUE
                                          }
                                }
                      }while(($password.length -lt $passwordLength) -or ($($containsLower -and $containsUpper -and $containsSpecial -and
$containsDigit) -eq $FALSE))
                      Write-Output $password
           }
           finally{
                      $random=$NULL
                      $byte=$NULL
           }
}

function updateHostPasswordsPS([ref]$machinesArrayRef, [ref]$passwordsArrayRef, $localAccountName, $outFilePath, $maxThreads){
           try{
                      #$outFilePath might not exist...can't use resolve-path.
                      $absOutFilePath = $ExecutionContext.SessionState.Path.GetUnresolvedProviderPathFromPSPath($outFilePath)
                      $errorPath = Join-Path (Split-Path $absOutFilePath -Parent) "machinesNotChanged.out"

                      Out-File -FilePath $absOutFilePath > $NULL #clear file
                      Out-File -FilePath $errorPath > $NULL #clear file

                      for($i=0; $i -lt $machinesArrayRef.value.length; $i++ ){
                                #iterate over all machine names and change password
                                $machineName = $($machinesArrayRef.value[$i])
                                $setPasswordSB = {
                                          #script block for threaded password changing
                                          #get local information through WinNT provider and change password
                                          #through iADSuser interface. Output successful pw changes to
                                          #$outFilePath and unsuccessful changes to $errorPath and console
                                          param($machineName, $ssPassword, $localAccountName, $outFilePath, $errorPath)

                                          function Append-FileThreadSafe($filepath, $data){
                                                    #appends $data to the file located at $filepath in a threadsafe
                                                    #manner. This script uses a global semaphore for mutual exclusion
                                                    $sem = New-Object -TypeName System.Threading.Semaphore(1,1,"Global\PWSem1")
                                                    [void]$sem.waitOne()
                                                    try{

                                                              Out-File -FilePath $filepath -InputObject $data -Append > $NULL
                                                    }
                                                    finally{
                                                              [void]$sem.release()
                                                              $sem = $NULL
                                                    }
                                          }
                                          function Format-ErrorMessage($errorMessage){
                                                    if($errorMessage){
                                                              $msg = $errorMessage.split(":")
                                                              Write-Output (($errormessage.split(":"))[1].replace('"', "").trim())
                                                    }
                                          }
                                          $user = $NULL

                                          try{
                                                    $user = [ADSI]"WinNT://$machineName/$localAccountName"

           $user.setPassword($([Runtime.InteropServices.Marshal]::PtrToStringAuto([Runtime.InteropServices.Marshal]::SecureStringToBStr($ssPa
ssword))))
                                                    if($?){
                                                              $user.setInfo() > $NULL
                                                              $output =
"$machineName`t`t$localAccountName`t`t$([Runtime.InteropServices.Marshal]::PtrToStringAuto([Runtime.InteropServices.Marshal]::SecureStringToB
Str($ssPassword)))"
                                                              Append-FileThreadSafe $outFilePath $output

                                                    }
                                                    else{
                                                              Append-FileThreadSafe $errorPath $machineName
                                                              $error | foreach{Format-ErrorMessage $_.Exception.message} | foreach{Write-
Host "Error changing password for $machineName`: $_"}
                                                    }
                                          }
                                          catch{
                                                    Append-FileThreadSafe $errorPath $machineName
                                                    $error | foreach{Format-ErrorMessage $($_.Exception.message)}  | foreach{Write-Host
"Error changing password for $machineName`: $_"}
```

```
                                        }
                                        finally{
                                                $user = $NULL
                                        }
                                }
                                while((Get-Job | where {$_.state -ne "Completed"}).count -ge $maxThreads){
                                        Get-Job | Wait-Job -Any -Timeout 5 > $NULL
                                        Get-Job -State "Completed" | Receive-Job
                                        Remove-Job -State "Completed"

                                }
                                Start-Job -Name $machineName -ScriptBlock $setPasswordSB -ArgumentList $machineName,
$passwordsArrayRef.value[$i], $localAccountName, $absOutFilePath, $errorPath > $NULL
                        }
                        #wait until all jobs are finished or 90 seconds, whichever comes first
                        Get-Job | Wait-Job -Timeout 90 > $NULL
                        #get the output
                        Get-Job | Receive-Job
                        try{
                                #clean up after ourselves
                                [void](Remove-Job * -force)
                        }
                        catch{
                                #suppress "powershell has already disposed of PSJob error"...
                        }
           }
           finally{$jobs = $NULL}
}

function Get-Volume($volumeLetter){
           $diskDrives = Get-WmiObject Win32_diskdrive
           foreach($drive in $diskDrives){
               #loop over each drive and see if it is attached via usb
            $partitions = $($drive.GetRelated('Win32_DiskPartition'))
            foreach($partition in $partitions){
                #loop over each partition on the disk
                if($partition){
                       $logicalDisks = $($partition.GetRelated('Win32_LogicalDisk'))
                       foreach($logicalDisk in $logicalDisks){
                           #each partition has a logical disk or volume associated with it
                           if($logicalDisk -and ($logicalDisk.DeviceID.startswith($volumeLetter))){
                                return $logicalDisk
                           }
                       }
                 }
            }
           }
}

function isPathOnUSBDrive($usbDrives, $path){
           #convert path to the containing directory of the absolute path
           $absPath = [IO.Path]::GetFullPath($outFilePath)
           if((Test-Path $path) -eq $FALSE){
                      #make sure the path exists
                      [console]::WriteLine("The path: '$path' does not exist")
                      throw "No such path exists"
           }
           else{
                      foreach($drive in $usbDrives){
                                #check to see if the path of the file starts with one of these USB drive letters
                                if($path.startswith($drive)){
                                        return $TRUE
                                }
                      }
           }
           return $FALSE
}

function Get-USBDrives(){
           $diskDrives = Get-WmiObject Win32_diskdrive
           foreach($drive in $diskDrives){
               #loop over each drive and see if it is attached via usb
               if($drive.InterfaceType -eq "USB"){
                   $partitions = $($drive.GetRelated('Win32_DiskPartition'))
                   foreach($partition in $partitions){
                       #loop over each partition on the disk
                       $logicalDisks = $($partition.GetRelated('Win32_LogicalDisk'))
                       foreach($logicalDisk in $logicalDisks){
                           #each partition has a logical disk or volume associated with it
                           Write-Output $logicalDisk.DeviceID
                       }
                   }
               }
           }
}

function volumeContainsSufficientSpace($outFilePath, $minFileSpace){
           $absPath = [IO.Path]::GetFullPath($outFilePath)
           $volume = Get-Volume $absPath.Drive
           return $volume.FreeSpace -gt $minFileSpace
}
```

```
function Main(){
	try{
		if($forceUSBKeyUsage -eq $TRUE){
			$usbDrives = @(Get-USBDrives)
			if($(isPathOnUSBDrive $usbDrives $outFilePath) -eq $FALSE){
				#abort program. The outfile path is not on a removable device
				[console]::WriteLine("The outfile path is not on a removable device")
				throw "The outfile path is not on a removable device"
			}
		}
		#see if volume contains enough space
		if((volumeContainsSufficientSpace $outFilePath 2MB) -eq $FALSE){
			throw "disk does not have enough free space to save password file"
		}
		$machinesArray = $NULL
		#if user specifies a machine file, use it. Otherwise, automatically detect
		#the machines registered to the domain.
		if($machinesFilePath -ne ""){
			$machinesArray = @(readInFile $machinesFilePath)
		}
		else{
			$machinesArray = @(Get-DomainComputersSansDomainControllers)
		}

		$passwordsArray = @(foreach($m in $machinesArray){
		$passwordLength = Get-Random -Minimum $minPasswordLength -Maximum ($maxPasswordLength+1);
			Write-Output (genPassword $passwordLength)
                                                                })
		if($machinesArray.length -ne $passwordsArray.length){
			#this really should be an assert, but powershell doesn't have assertions...
			throw "machinesArray($machinesArray.length)and passwordsArray ($passwordsArray.length) are not equal"

		}
		updateHostPasswordsPS ([ref]$machinesArray) ([ref]$passwordsArray) $localAccountName $outFilePath $maxThreads

	}
	finally{
		#clean up SecureString objects
		$passwordsArray | foreach{$_.dispose()}
	}
}


Main
```

Appendix B: Denying local administrators network access

This script scans the local computer for any local administrator accounts. Domain accounts that have local admin privileges are ignored. It creates a local group with a well-known name, which is **DeniedNetworkAccess** by default, and then adds the local administrator accounts to the local group. The domain administrator then adds the well-known group name to the domain policy so that it is part of the **Deny access to this computer from the network** and **Deny log on through Remote Desktop Services** (for Windows Server 2008R2 and later) policies under **Computer Configuration** > **Windows Settings** > **Security Settings** > **Local Policies** > **User Rights Assignment**. The script should be deployed as a Group Policy Computer Startup or Shutdown script.

It is very important to note that the group name created by this script should not be added to the GPO policy for denying network access until the group has actually been created by the script. If the Group Policy has been configured, but the group does not exist on the system, then a Group Policy error will occur. Before configuring the policy, deploy the script as a startup script and force systems to reboot before adding the group name to the policy.

```
Option Explicit

' This is the name of the group that will be added to the 'Denied access to this computer from the network' policy setting.
Const GROUP_NAME = "DeniedNetworkAccess"
```

```
Call Main()


Sub Main
   Dim computerName, domainName, currentDeniedUsers, adminGroupName, administrators, localAdministrators, account

   computerName = GetComputerName()
   domainName = GetComputerDomainName()

   If Not LocalGroupExists(GROUP_NAME, computerName) Then
      Call CreateLocalGroup(GROUP_NAME, computerName)
   Else
      ' if the group exists then remove all current group members before we add members later on
      ' this prevents the group membership from getting out of sync since we do not want to always just add users to the list
      Call RemoveAllUsersFromLocalGroup(GROUP_NAME, computerName)
   End If

   ' get the name of the local admin group since it could possibly be renamed
   adminGroupName = GetLocalAdministratorsGroupName(computerName)

   ' get all the users who are currently in the local admin group
   Set administrators = GetAllUsersFromLocalGroup(adminGroupName, computerName)

   ' do not mess with any domain accounts in the local admin group, so only get the local accounts from the list
   Set localAdministrators = GetLocalUsersFromUserList(administrators, computerName, domainName)

   For Each account in localAdministrators.Keys
      Call AddUserToLocalGroup(localAdministrators.Item(account), GROUP_NAME, computerName)
   Next

End Sub


' Gets the name of the local computer.
Function GetComputerName
   Dim network, name

   Set network = CreateObject("WScript.Network")
   name = network.ComputerName
   Set network = Nothing

   GetComputerName = LCase(CStr(name))
End Function


' Gets the domain name the local computer is joined to.
Function GetComputerDomainName
   Dim network, domain

   Set network = CreateObject("WScript.Network")
   domain = network.UserDomain
   Set network = Nothing

   GetComputerDomainName = LCase(CStr(domain))
End Function


' Gets the name of the local admin group since it could be renamed.
Function GetLocalAdministratorsGroupName(computerName)
   Dim wmi, groups, group, name

   Set wmi = GetObject("winmgmts:\\" & computerName & "\root\cimv2")

   Set groups = wmi.ExecQuery("Select Domain, Name, SID From Win32_Group Where Domain = '" & computerName & "' and SID='S-1-5-32-544'")

   For Each group in groups
      name = CStr(group.Name)
      Exit For
   Next

   Set group = Nothing
   Set groups = Nothing
   Set wmi = Nothing

   GetLocalAdministratorsGroupName = name
End Function


' Returns True if the specified local computer group exists, otherwise it returns False.
Function LocalGroupExists(groupName, computerName)
   Dim wmi, groups, groupExists

   Set wmi = GetObject("winmgmts:\\" & computerName & "\root\cimv2")

   ' WMI on Windows 2000 could have a problem when using both Domain and Name properties in the Where clause. See MS KB268715
   Set groups = wmi.ExecQuery("Select Domain, Name From Win32_Group Where Domain = '" & computerName & "' And Name='" & groupName & "'")

   If groups.Count >= 1 Then
      groupExists = vbTrue
   Else
      groupExists = vbFalse
   End If
```

```
   Set groups = Nothing
   Set wmi = Nothing

   LocalGroupExists = groupExists
End Function


' Creates the specified local computer group.
Sub CreateLocalGroup(groupName, computerName)
   Dim computer, group

   Set computer = GetObject("WinNT://" & computerName & ",Computer")

   Set group = computer.Create("group", groupName)
   group.SetInfo

   Set group = Nothing
   Set computer = Nothing
End Sub


' Gets all users that are in a local group.
' Results are returned in a dictionary with the key as the user name and the item as the ADsPath.
Function GetAllUsersFromLocalGroup(groupName, computerName)
   Dim users, group, member

   Set users = CreateObject("Scripting.Dictionary")

   Set group = GetObject("WinNT://" & computerName & "/" & groupName & ",Group")

   For Each member in group.Members
      users.Add CStr(member.Name), CStr(member.ADsPath)
   Next

   set group = Nothing

   Set GetAllUsersFromLocalGroup = users
End Function


' Filters out any domain users accounts from a list of user accounts.
' The function expects the list to be a dictionary with the key as the user name and the item as the ADsPath.
' Results are returned in a dictionary with the key as the user name and the item as the ADsPath.
Function GetLocalUsersFromUserList(users, computerName, domainName)
   Dim user, localAccounts

   Set localAccounts = CreateObject("Scripting.Dictionary")

   For Each user in users.Keys
      If InStr(1, LCase(users.Item(user)), LCase("//" & domainName & "/" & computerName), 1) >= 1 Then
         localAccounts.Add user, users.Item(user)
      Else
         ' skip this account because it is a domain account
      End If
   Next

   Set GetLocalUsersFromUserList = localAccounts
End Function


' Adds a set of users to the specified local group for the specified computer.
Sub AddUserToLocalGroup(user, groupName, computerName)
   Dim group

   Set group = GetObject("WinNT://" & computerName & "/" & groupName & ",Group")

   group.Add user

   Set group = Nothing
End Sub


' Removes all members from a local group
Sub RemoveAllUsersFromLocalGroup(groupName, computerName)
   Dim group, member

   Set group = GetObject("WinNT://" & computerName & "/" & groupName & ",Group")

   For Each member in group.Members
      group.Remove(member.ADsPath)
   Next

   group.SetInfo
   set group = Nothing
End Sub
```

Appendix C: Configuring Windows Firewall rules

This section explains how to use Active Directory to deploy Windows Firewall rules to help mitigate PtH. By creating a set of firewall rules that disallow incoming connections to workstation computers on a Windows domain, attackers will not be able to spread laterally using PtH to workstations which may be less monitored and have a higher attack surface than servers.

The following guidance explains how to define and deploy Windows Firewall rules that block workstation to workstation communication. The domain controller must be running Windows 2008 R2 and later and the workstations must be running Windows Vista and later.

1. Create a new GPO called **pthGPO.**
2. Right-click the **pthGPO** GPO and select **Edit**. The Group Policy Management Editor dialog box will appear.
3. Navigate to **Computer Configuration > Policies > Windows Settings > Security Settings > Windows Firewall with Advanced Security > Windows Firewall with Advanced Security – LDAP://CN=xxxx**.
4. Right click on **Firewall with Advanced Security – LDAP://CN=xxxx** and click on **Properties**.
5. In this dialog box there are three profiles: domain, private, and public. For each profile, in the **State** section:
    a. Set the **Firewall state** to **On**, the **Inbound connections** to **Block (default)**, and **Outbound Connections** to **Not Configured**.
    b. In the **Settings** box, click the **Customize** button.
    c. In the **Rule merging** section, set the **Apply local firewall rules** drop down to **No**.
    d. In the **Rule merging** section, set the **Apply local connection security rules** drop down to **No**.
6. Click **OK** to finish the configuration.
7. Within the Group Policy Management Editor, navigate to **Computer Configuration > Policies > Windows Settings > Security Settings > Windows Firewall with Advanced Security > Windows Firewall with Advanced Security – LDAP://CN=xxxx**,right-click on **Inbound Rules**, and then click **New Rule**.
8. Select the **Custom** radio button and click **Next**.
9. Select the **All programs** radio button and click **Next** three times.
10. In the **Which remote IP addresses does this rule apply to?** section, select the **These IP address or subnet** radio button.
11. Click the **Add** button and specify the IP address of the system(s) allowed to make inbound connections to the client workstations. The values entered in Figure 1 are specifically tailored to the requirements of each network's infrastructure and services. [6] lists the port requirements for the common Microsoft services. If a service's port requirements are not listed, then see its documentation. In Figure 1, the IP address is the domain controller.

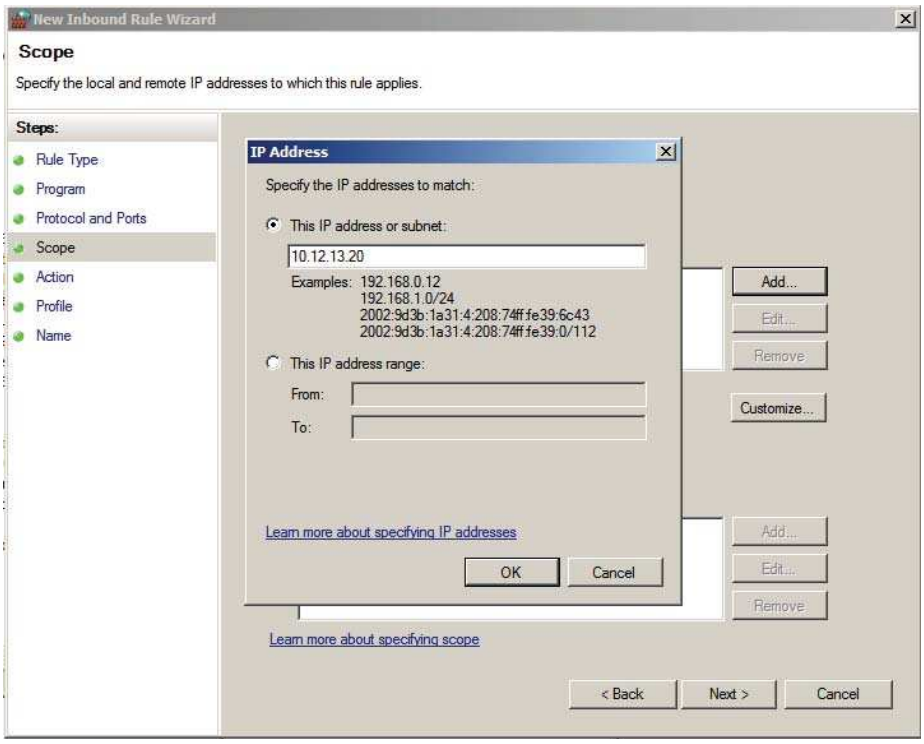

Figure 1: Scope dialog box.

12. Click **OK** and then **Next**.
13. Select the **Allow this connection** radio button and click **Next**.
14. Make sure **Domain**, **Private**, and **Public** boxes are all checked and click **Next**.
15. Click **Finish.**
16. Within the **Group Policy Management** console navigate to **Forest > Domains**, right-click the domain name, and select **New Organizational Unit**. A dialog box will appear.
17. Type in the name of the new OU. In this case, the new OU is named **pthOU**. Click **OK.**
18. Right-click **pthOU** and select **Link an Existing GPO**. The **Select GPO** dialog will appear.
19. Select **pthOU** from the list of GPOs and click **OK**.
20. In the **Security Filtering** section of the pthGPO shown in Figure 2, click the **Add** button.

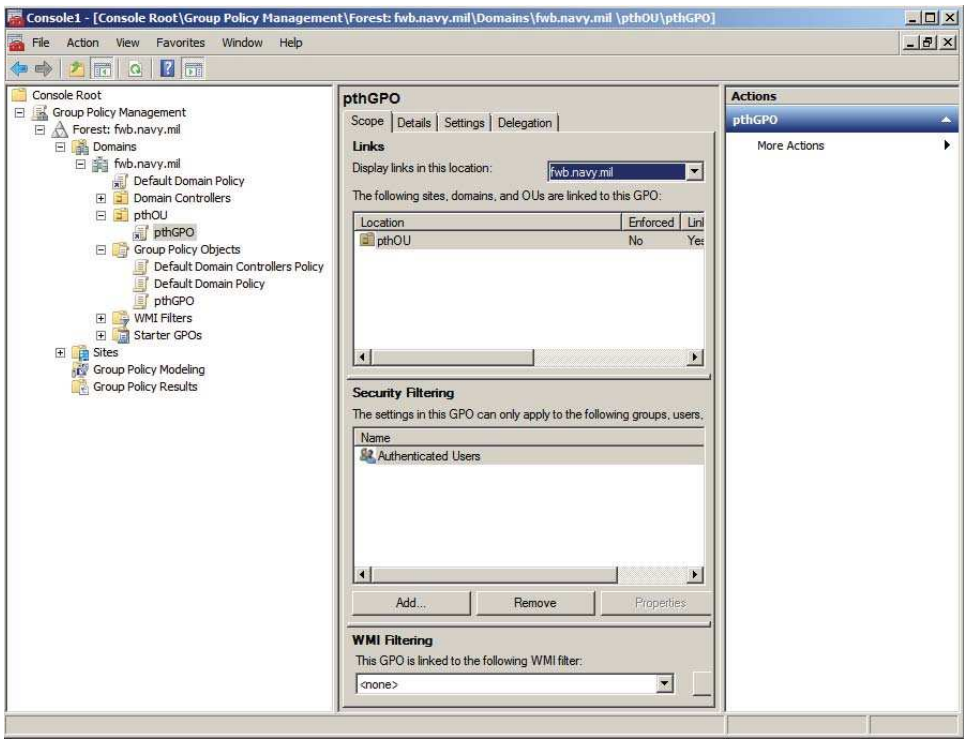

Figure 2: The new GPO linked to the PtH Organization Unit.

21. Click the **Object Types** button and make sure that the **Computer** option is selected as shown in Figure 3. Click **OK** when finished.

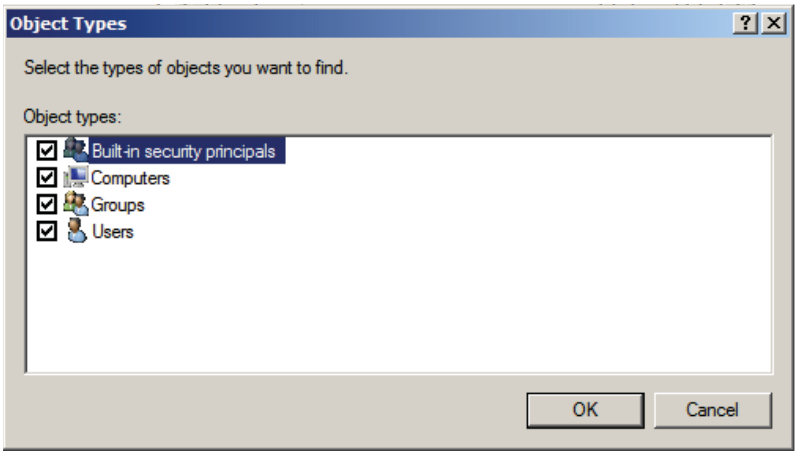

Figure 3: Object types dialog box settings.

22. In the **Enter the object name to select box** type in the name of the system to be subject to the GPO. Click **Check Names** to verify the name. Click **OK**.
23. Repeat the two previous steps for each computer subject to the GPO.
24. Reboot each of the clients or issue the **gpupdate /force** command from each client.